Chapitre 3
L'indexation avec MongoDB

1. Comment ça marche ?

Dans la terminologie des bases de données, relationnelles ou non, un index est en tous points semblable à celui que l'on trouve à la fin de n'importe quel livre : on y regroupe les termes importants qui figurent dans l'ouvrage avec, en face de ceux-ci, les numéros des pages dans lesquelles ils se trouvent. Ceci nous évite tout simplement d'avoir à relire la totalité du livre lorsque nous cherchons un simple terme.

Par analogie, un index posé sur le champ ou le sous-champ d'une collection nous évite d'avoir à parcourir toute notre collection pour retrouver les valeurs de ce champ (ou sous-champ) correspondant à notre requête et participe ainsi à garder le temps d'exécution de nos requêtes le plus petit possible.

Les index ont des avantages et des inconvénients : ils améliorent considérablement les temps d'exécution des requêtes en lecture, mais ils ralentissent les opérations d'écriture telles que les insertions, les suppressions ou les mises à jour, qui nécessitent leur reconstruction. Cependant, les ralentissements observés sont généralement négligeables par rapport à la réduction du temps d'exécution qu'ils engendrent.

Pour savoir quels champs d'une collection ont besoin d'être indexés, il faut avoir une idée des requêtes sur celle-ci et de leur fréquence. Des champs qui sont souvent ciblés par des requêtes devront être indexés en priorité. Un site web qui propose un moteur de recherche de produits sous la forme d'une zone de saisie de texte devra avoir une collection produits dans laquelle le nom des produits ou encore la catégorie à laquelle ils appartiennent sont indexés, car les utilisateurs y entrent majoritairement des noms de produit (« iPhone 10 ») ou bien des noms de catégories (« cafetières italiennes »).

La nature de notre applicatif va directement impacter notre logique d'indexation : est-elle plutôt orientée écriture (*write-heavy*) ? Plutôt lecture (*read-heavy*) ? Un site web marchand n'est-il pas à la croisée des chemins ? En effet, on y fait de nombreuses lectures (recherches de produits, campagnes d'e-mailing promotionnel depuis la base clients, API mises à disposition de partenaires), mais aussi des écritures (paniers, commandes, approvisionnement de notre catalogue de produits, mises à jour quotidiennes des prix en fonctions de critères fournisseurs), dès lors vous comprenez bien qu'on ne peut pas décider d'une stratégie d'indexation sur un coin de table.

2. Index simples

Lorsqu'une collection est créée, MongoDB génère automatiquement un index sur le champ `_id`. Cet index ne peut aucunement être supprimé, car il garantit l'unicité même de cet identifiant. Pour créer vous-même un index, il vous faut utiliser la fonction `createIndex` dont voici la syntaxe :

```
db.collection.createIndex(< champ_et_type >, < options >)
```

Laissez-moi vous présenter la nouvelle version de notre collection `personnes` :

```
db.personnes.drop()

db.personnes.insertMany(
[
{"nom": "Durand", "prenom": "René", "interets": ["jardinage",
"bricolage"], "age": 77},
{"nom": "Durand", "prenom": "Gisèle", "interets": ["bridge",
"cuisine"], "age": 75},
{"nom": "Dupont", "prenom": "Gaston", "interets": ["jardinage",
"pétanque"], "age": 79},
```

```
{"nom": "Dupont", "prenom": "Catherine", "interets": ["cuisine"], "age": 66},
{"nom": "Duport", "prenom": "Eric", "interets": ["cuisine",
"pétanque"], "age": 57},
{"nom": "Duport", "prenom": "Arlette","interets": ["jardinage"], "age": 80},
{"nom": "Lejeune", "prenom": "Jean","interets": ["jardinage"], "age": 75},
{"nom": "Lejeune", "prenom": "Mariette","interets": ["jardinage", "bridge"],
"age": 66}
]
)
```

Créons sans plus tarder un premier index sur le champ `age` des documents de la collection. Lorsque l'on crée un index, il est obligatoire de mentionner l'ordre d'apparition des valeurs de ce champ dans l'index : cet ordre est croissant ou décroissant (respectivement 1 et -1, exactement comme pour `sort()`). Nous avons décidé que notre index sur le champ `age` contiendra les valeurs prises par `age`, triées par ordre décroissant :

```
db.personnes.createIndex({"age": -1})
```

Pour valider que la création de cet index est conforme aux attentes, nous pouvons également utiliser la fonction `getIndexes` en l'appliquant à notre collection :

```
db.personnes.getIndexes()
```

Celle-ci nous affiche bien un tableau contenant deux documents avec les détails de nos index :

```
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { age: -1 }, name: 'age_-1' }
```

Vous voyez s'afficher le champ sur lequel est posé l'index, l'ordre d'indexation (croissant ou décroissant) ainsi que le nom donné à notre index par MongoDB. Par défaut, notre index porte le nom du champ ciblé auquel est concaténé un caractère souligné suivi de l'ordre, ce qui nous donne ici `age_-1`. Ce n'est ni très esthétique ni très parlant, voilà pourquoi nous allons le nommer de façon différente.

Pour réaliser cette opération de renommage d'index, nous allons commencer par supprimer l'index existant avant d'en recréer un avec un nouveau nom : `idx_age`. La suppression d'un index s'effectue avec la commande `dropIndex` à laquelle nous passerons l'ancien nom de l'index.

Le nom de notre nouvel index sera quant à lui spécifié dans le document qui contient les paramètres de l'index, en face de la clé `name`. Pour conclure, nous listerons les index de notre collection pour vérifier la bonne création de cet index portant le nouveau nom.

```
db.personnes.dropIndex("age_-1")
db.personnes.createIndex({"age": -1}, {"name": "idx_age"});
db.personnes.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { age: -1 }, name: 'idx_age' }
]
```

Notre collection exemple est d'une taille plus que modeste et nous sommes seuls à l'utiliser, aussi nous pouvons nous permettre de reconstruire l'index en le supprimant. Vous pouvez imaginer sans peine que l'impact serait tout autre sur une collection comptant plusieurs centaines de milliers (voire millions !) de documents qui est utilisée par tout le personnel d'une grande entreprise. Il conviendrait alors d'exécuter ces différentes procédures dans une fenêtre de temps propice à la maintenance (en pleine nuit par exemple) et en utilisant l'option `background` pour signifier que cette tâche est une tâche de fond, de moindre priorité.

Par exemple, pour créer un index classé par ordre alphabétique sur le champ `prenom` en tâche de fond, vous écrirez :

```
db.personnes.createIndex( { "prenom": 1 }, { "background": true})
```

Lors de la création d'un index comme lors de la création d'une collection ou d'une vue (que nous verrons plus tard), il est possible de préciser une collation, mais attention, cette fonctionnalité n'est disponible que depuis la version 3.4 de MongoDB. Si nous reprenons l'index précédent en lui ajoutant les options relatives à la nature de la collation, nous obtenons :

```
db.personnes.createIndex(
  { "prenom": 1 },
  { "background": true,
  "collation": {
     "locale": "fr"
  }
 }
)
```

Lorsqu'un index simple est créé avec une collation, seules les requêtes qui précisent cette même collation pourront s'appuyer dessus, autrement une opération de *collection scan* sera effectuée, car elles utilisent la comparaison binaire par défaut. Prenons ce dernier index que nous venons de créer. La requête suivante l'utilisera :

```
db.personnes.find({"prenom": "René"}).collation({locale: "fr"})
```

Tandis que la suivante ne pourra pas s'appuyer dessus :

```
db.personnes.find({"prenom": "René"})
```

3. Index composés

Un index peut porter sur plus d'un champ : c'est ce que l'on appelle un index composé (*compound index*). Dans ce type d'index, l'ordre dans lequel les champs sont énumérés a son importance. Supprimons notre index `idx_age` et créons un index composé nommé `idx_age_nom` qui portera sur l'âge puis sur le nom des personnes :

```
db.personnes.createIndex({"age": 1, "nom": 1},
{"name": "idx_age_nom"})
```

L'index sera ordonné d'abord par valeurs croissantes d'âge et par ordre alphabétique de nom ensuite, au sein de chacune des différentes valeurs d'âge.

Lorsqu'un index composé dont le préfixe n'est pas une chaîne de caractères, un tableau ou un sous-document est utilisé avec une collation, une requête n'utilisant pas la bonne collation pour le champ texte indexé peut toutefois s'appuyer sur le préfixe de l'index.

Supposons que notre index précédent ait été créé de cette façon :

```
db.personnes.createIndex(
  {"age": 1, "nom": 1},
  {"name": "idx_age_nom", "collation": { locale: "fr" }}
)
```

La requête suivante qui utilise pourtant la collation binaire de base pour la comparaison des chaînes de caractères (et non `fr`) pourra néanmoins utiliser `idx_age_nom` car age en constitue le préfixe :

```
db.personnes.find({"age": {$gt: 40}, "prenom": "Christophe"})
```

Préfixe d'un index

Une requête peut utiliser la totalité des champs constituant l'index composé ou bien une sous-partie seulement, à la condition qu'elle soit constituée de champs figurant au début de l'index. Cette sous-partie de l'index est appelée le *préfixe* et on retrouve cette notion dans certains systèmes de gestion de bases de données relationnelles sous le nom de *leftmost prefix* (littéralement « préfixe le plus à gauche »).

Dans le cas présent, notre index porte sur deux champs et il sera utilisé par les requêtes ciblant :

– le champ `nom` tout seul (champ le plus à gauche, débutant l'index et constituant son *préfixe*) ;
– les champs `nom` et `age` (c'est-à-dire l'index dans sa totalité).

Une requête ciblant des valeurs du champ `age` seulement ne bénéficiera pas de cet index composé, car ce champ ne fait pas partie du *préfixe*.

Écrivons un index composé nommé `idx_nom_prenom_age` qui va s'appliquer sur le triplet `nom`, `prenom` et `age` (dans cet ordre) :

```
db.personnes.createIndex(
  {"nom": 1,"prenom":1, "age": 1},
  {"name": "idx_nom_prenom_age"}
)
```

Vont pouvoir s'appuyer dessus :

– les requêtes portant sur le premier champ du *préfixe*, nom :

```
db.personnes.find({"nom": "Lejeune"})
```

– les requêtes portant sur les deux champs constituant le *préfixe*, `nom` et `prenom` :

```
db.personnes.find({"nom": "Lejeune", "prenom": "Jean"})
db.personnes.find({"prenom": "Jean", "nom": "Lejeune"})
```

– les requêtes portant évidemment sur la totalité des champs qui constituent l'index composé :

```
db.personnes.find({"nom": "Lejeune", "prenom": "Jean", "age": 75})
db.personnes.find({"nom": "Lejeune", "age": 75, "prenom": "Jean"})
db.personnes.find({"age": 75, "nom": "Lejeune", "prenom": "Jean"})
db.personnes.find({"age": 75, "prenom": "Jean", "nom": "Lejeune"})
```

– les requêtes contenant le *préfixe* ainsi qu'un autre champ :

```
db.personnes.find({"age": 75, "nom": "Durand"})
```

Par contre, les requêtes suivantes ne pourront pas s'appuyer sur notre index triple :

– celles qui ciblent un champ qui ne fait pas partie du *préfixe* :

```
db.personnes.find({"age": 75})
```

– celles qui font usage d'une partie seulement du *préfixe* (constitué rappelons-le des champs `nom` et `prenom`) :

```
db.personnes.find({"prenom": "Jean"})
db.personnes.find({"prenom": "Jean", "age": 75})
```

Les index peuvent également être sollicités lors du tri d'un curseur avec la méthode `sort` que nous avons vue précédemment. Si, dans le cas des index simples, l'ordre croissant ou décroissant n'a pas d'incidence sur le tri parce que MongoDB est capable de parcourir l'index dans n'importe quel sens, il n'en va pas de même pour les index composés : les champs utilisés dans `sort` doivent apparaître dans le même ordre que celui de l'index.

Créons un index composé `idx_age_nom` défini de la façon suivante :

```
{"age": 1, "nom": 1}
```

Cela signifie que les tris suivants vont l'utiliser :

```
db.personnes.find().sort({"age": 1})
db.personnes.find().sort({"age": 1, "nom": 1 })
```

Alors que les tris suivants ne pourront pas s'appuyer dessus :

```
db.personnes.find().sort({"age": 1})
db.personnes.find().sort({"age": 1, "nom": 1 })
```

```
db.personnes.find().sort({"nom": 1, "age": 1})
db.personnes.find().sort({"nom": 1})
```

Pour utiliser un index lors d'un tri, il faut également tenir compte de l'ordre spécifié lors de la création de celui-ci, c'est celui-ci et son exact inverse qui utiliseront l'index. Reprenons notre index composé `idx_nom_age` :

```
{"nom": 1, "age": 1}
```

Son inverse sera donc :

```
{"nom": -1, "age": -1}
```

Les tris qui utiliseront notre index composé seront les suivants :

```
db.personnes.find().sort({"age": 1, "nom": 1 })
db.personnes.find().sort({"age": -1, "nom": -1 })
```

Alors que ceux-ci ne pourront pas s'appuyer dessus :

```
db.personnes.find().sort({"age": -1, "nom": 1 })
db.personnes.find().sort({"age": 1, "nom": -1 })
```

En règle générale, lorsque MongoDB utilise un index dans une requête, il renvoie les documents dans l'ordre dans lequel l'index les a rangés. Prenons la requête suivante, qui cible les documents dont le nom commence par « Du » (c'est une *expression régulière* contenue entre deux barres obliques, notez juste qu'elle est sensible à la casse des caractères) :

```
db.personnes.find({"nom": /^Du/}, {"_id": 0, "nom": 1, "age": 1})
```

Avec notre index `idx_nom_age` sous la forme `{"nom": 1, "age": 1}`, voici ce que nous obtenons :

```
{ "nom" : "Dupont", "age" : 66 }
{ "nom" : "Dupont", "age" : 79 }
{ "nom" : "Duport", "age" : 57 }
{ "nom" : "Duport", "age" : 80 }
{ "nom" : "Durand", "age" : 75 }
{ "nom" : "Durand", "age" : 77 }
```